Code Red Robotics Crescendo Visions Docs

Gene Wicaksono, Elijah Kelly, Landon Cheung, Skye Ford,
Bradford Smith, Zachary Stillman

2024

2024 Visions Github (https://github.com/CRRobotics/2024Visions)

Contents

Abstract

This year, Code Red Robotics used computer vision for multiple tasks. Based on AprilTags on the field, we can find the robot's position on the field, which can allow for things like moving somewhere automatically. We also can use AprilTags to find the angle from the robot to the speaker, allowing automatic aiming. Another of the main things that Visions did this year was to implement note detection code. This code finds notes, locates them on the field, and then allows the driver to press a button to automatically drive to and acquire them.

1 Cameras and Computers

This year, we had two main Visions systems, which used different cameras and computers. AprilTag detection was done on a small-formfactor Intel PC running Linux and connected to 6 cameras, and Note Detection was done on the LimeLight camera and its built-in computer.

1.1 AprilTag Cameras

Last year we used 3 AprilTag cameras. This was because knowing our pose was most important while scoring, where all the April tags are in the same direction. This year, the AprilTags are located in a myriad of locations, and facing many different directions.

To complement this, we have doubled the number of cameras we had last year. We are now using 6 ELP megapixel USB cameras running up to 30fps. We multithread the pose estimation, with each camera running its own thread. When running, we average at around 85% CPU usage.

For mounting the cameras, we 3d-printed mounts that are bolted on to the superstructure of our robot. Each camera is mounted with a spacer, and a cover, and is tiled with a pitch of 25 degrees. This pitch is critical, as it helps our robot determine pose when very close to the amp, speaker, and trap.

1.2 The LimeLight

This year, we used the LimeLight Camera for note detection. This is a camera designed for FRC, and although we're not using its built-in green LEDs, it has many other useful features. The main advantage is that it's a smart camera capable of storing and running Python code, which it automatically provides with the image it sees at each frame. It also automatically pushes values we give it to NetworkTables, which allows them to be accessed by Programming's code.

LimeLight Documentation
(https://docs.limelightvision.io/docs/docs-limelight/getting-started/summary)

2 Note Detection

Each year in FIRST, game piece detection is a useful vision feature to have on a robot. There are multiple ways to do it, including AI, but we decided to use an HSV filter, followed by finding contours. The contours then have ellipses fit to them, which has multiple advantages. After the detection process, the code has a list of potential notes to find the coordinates of.

2.1 HSV Filtering

The first step in detecting notes was to filter for anything orange. To do this, we made an HSV threshold filter. HSV stands for Hue, Saturation, Value, which is a format used for representing colors. In HSV, colors are broken down into:

- Hue: an angle representing where the color is on the color wheel
- Saturation: a value representing how pure the color is, or how far from grayscale
- Value: the brightness of the color

By setting upper and lower bounds for each of these parameters, we were able to filter pretty accurately for the orange color of the note. We did this with OpenCV's `inRange` function:

```
mask = cv2.inRange(image, lower, upper)
```

Of course, the filter needs to be adjusted each time it's used depending on the lighting of the room. Also, it can be hard to distinguish orange from red, or, surprisingly, light skin. To help filter out things that are just on the edge of the color filter, such as red, we also added a median blur. This is a kind of blur that sets each pixel to the median of the surrounding pixels, which reduces noise caused by outliers. By blurring the image before filtering, we could get rid of noise that could cause the code to see something that's not orange.

```
blurred = cv2.medianBlur(image, BLUR_SIZE * 2 + 1)
mask = cv2.inRange(blurred, lower, upper)
```

By doing these steps, we were able to produce a filter that pretty accurately finds orange objects.

One thing that was sometimes needed for the HSV filter was to have an inverted hue range. Because orange is close to red, we needed to be able to detect red in some cases, and red is present at both very low and very high hue values. To detect hue at low and high areas, we made an `invertedHueMask` function. This method takes hues above a high value and below a low value, and then combines these.

```
def invertedHueMask(image):
    """Returns an HSV mask with the hue range inverted,
    similar to what LimeLight's built-in color threshold
    can do. Likely won't be needed."""
    lower1 = np.array([0, lower[1], lower[2]])
    upper1 = np.array([lower[0], upper[1], upper[2]])
    lower2 = np.array([upper[0], lower[1], lower[2]])
    upper2 = np.array([HUE, upper[1], upper[2]])
    mask1 = cv2.inRange(image, lower1, upper1)
    mask2 = cv2.inRange(image, lower2, upper2)
    return cv2.bitwise_or(mask1, mask2)
```

2.2 Contours and Ellipses

Anything that gets past our color filter we fit a contour to. Then we fit ellipses to the contours to represent edge of the note, using `cv2.fitEllipse`. However, these ellipses do not accurately represent the note, as they go from the bottom part of the front to the top part of the back. The center of the note is an ellipse, but that's not what the camera sees due to the thickness. To compensate for this, we used a method that, although we have not proved to work, seems to work pretty well.

The code starts with an ellipse fit to the shape of the note. The major axis of this will be along the part of the note perpendicular to the camera axis. Because the note diameter is 14 inches and the thickness is 2 inches, the center ellipse should have a 12-inch diameter. The parts of the note along the major axis should be around the same distance from the camera, and represent the average thickness that the camera sees. To get the ellipse to the center of the note on the sides, we scale the major axis by 12/14. We then calculate now many pixels this subtracts, and subtract the same amount from the minor axis. This creates an ellipse that is on average in the center of the note torus, which should approximate the center ellipse of the note. The code for this is in our `fitEllipsesToNotes` function:

```
    ellipse = cv2.fitEllipse(hull)
    newMajor = ellipse[1][1] * (1 - NOTE_THICKNESS_IN\
            / NOTE_OUTER_RADIUS_IN) # Shrink the ellipse to be at
            # roughly the center of the torus
    newMinor = ellipse[1][0] - ellipse[1][1]\
            * NOTE_THICKNESS_IN / NOTE_OUTER_RADIUS_IN # Removing the
            # same amount from the minor axis as the major axis
    ellipse = list(ellipse) # Tuples must be
            # converted to lists to edit their contents
    ellipse[1] = (newMinor, newMajor)
    ellipse = tuple(ellipse)
```

Ellipse fitting was originally intended to calculate the position of the note relative to the robot using trig, but it turned out that the best method simply

looks at the center of the note. However, ellipse fitting still has some advantages, a large one being that if the note is partially obscured, fitting an ellipse to it could still give a better approximation of its center than simply finding the center of the contour.

3 Note Location

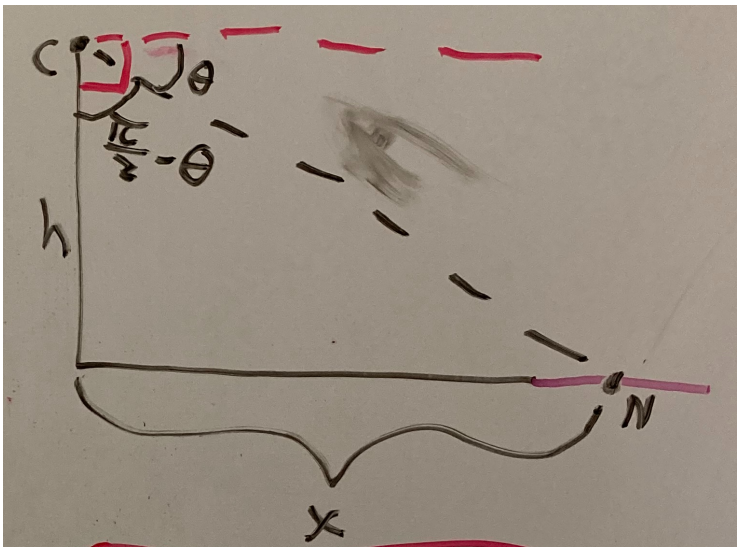
3.1 Pixels to Inches

Once we have center points, we need to convert their pixel coordinates on the camera plane into inch coordinates relative to the robot. To do this, we start by converting pixels to angles. This can be achieved by finding the horizontal and vertical FoV of the camera in both degrees and pixels. Once we have these, we can create conversion fractions for pixels to degrees in both the horizontal and vertical directions. For ease of computing, we then convert this to radians.

These angles are the horizontal and vertical optical angles of the point relative to the camera. Assuming that this point is the center of the note which is 1 inch above the ground, we can use these angles to calculate the note's ground angles relative to the robot. We initially tried to do this ourselves, but it turned out that our 2022 Vision subteam faced a nearly identical problem with finding the coordinates of a hoop to shoot a piece into. We were hence able to adapt their math to our purposes, as well as figure out the math they did in more detail. The 2022 documentation can be found here (https://docs.google.com/document/d/1MrPBMtFBkQ2hRZHdlyit3DcHpTZ7xZSn6OE-ejCrV3k/edit?usp=sharing).

Once we have the ground horizontal and vertical angles, we can find the coordinates of the note. Using trig based on the height of the robot, we can find the distance to the note based on the vertical angle[1]:

[1] We also came up with 2 other methods of finding the distance. The first involved finding the angle that the note was tilted away from the camera based on the ellipse fit to it, and based on that and the assumption that it is on the floor, getting its distance. The second involved finding the optical angle that the note takes up in the camera FoV, and converting that to a distance using trig. However, we found through testing that this method worked the best, likely because it has the least number of potentially-error-introducing steps.



$$x = h\tan(\frac{\pi}{2} - \theta) = h\cot(\theta) = h/\tan(\theta)$$

We also have the ground angle, but this is from the forward direction to the note. To change this to the polar θ, we subtract the ground angle from $\frac{\pi}{2}$. We now have a distance and angle, which make up polar coordinates.

In the rectangular coordinate system we're using, x is the horizontal axis relative to the robot, and z is the axis that points forward. We can convert the polar coordinates to x and z coordinates, which are a bit easier for Programming to work with and for us to test.

3.2 More Filtering

Once we have found the coordinates of objects the code thinks are notes, we need to narrow them down to the one most likely to actually be a note. We originally tried to do this by figuring out the note's radius, but this turned out to not be reliable enough. We then decided on another method, which turned out to work much better.

A note we want to pick up will likely be the closest one to the camera, which would also, in theory, create the largest contour on the camera image. So, we want to find the object that is closest and has the largest contour. These don't always correspond with each other, so we wrote a function that factors in both by dividing the size by the distance, storing this in `sizeDistance`, and picking the contour with the greatest `sizeDistance`. This means that closer objects and bigger objects are more likely to be picked. Because the LimeLight cannot see directly under it, distance should not be 0, but we still wanted to make sure we don't accidentally get an error, so we just set `sizeDistance` to 0 (a very unappealing value) if distance is 0.

```
def closestNote(contours, distances, angles, ellipses):
    """Finds the closest and biggest contour."""
    if len(ellipses) == 0:
        return [], [], [], []
    maxNote = 0
    maxIndex = -1
    for i in range(len(ellipses)):
        contour = contours[i]
        distance = distances[i]

        if distance == 0: sizeDistance = 0
        else: sizeDistance = cv2.contourArea(contour) / distance
        if sizeDistance > maxNote:
            maxNote = sizeDistance
            maxIndex = i
    return [contours[maxIndex]], [distances[maxIndex]],\
            [angles[maxIndex]], [ellipses[maxIndex]]
```

Once we have decided on a note, we can send its coordinates to Programming's code via the NetworkTables functionality built into the LimeLight. If no note is detected, `closestNote` will return empty arrays, so 0 is appended to the coordinate arrays before they are returned. This means that if no note is detected, Programming will receive zeros for the coordinates of the note, which their code will interpret as "no note detected."

3.3 Testing

To verify that our code worked, we had to do a lot of testing. This involved putting the camera in various positions and having notes in its FoV. Based on these tests, we were able to find problems in our code and determine its accuracy. Our spreadsheet can be found here:

Testing Spreadsheet (https://docs.google.com/spreadsheets/d/1CyS7Q9FoU-tGURZ5oeh2BVDFtO9mNx1mUbKxktvOXEQ/edit#gid=377547610)

One thing that we found interesting is that over all of our tests, we found clues that there was some internal problem with the camera. One of these was that we used multiple different methods for calculating coordinates of a note, and all of them, even ones that had worked in the past, had a similar error. As well as this, we noticed that error increased a lot in a specific region of the camera. Lastly, we found that the center of the camera image did not match up with what appeared to be the center in reality.

To remedy this, we attempted calibrating the camera. However, our reprojection error was nearly 50, which is very bad considering that a calibration is considered good if the reprojection error is less than 1. The thing is, we did not really see much distortion in the calibrated image, suggesting that our calibration worked as well, if not better, than the calibration the camera was previously using. The error still remained, though.

In the end, we just decided that our error was within the acceptable range, but we may end up trying to manually correct for it. After testing with the actual robot, we found that our code was able to very reliably pick up notes, especially if the driver first drove within a reasonable distance to them. The main problem occurs if there is something else red or orange on the field, but due to our filter for the biggest and closest object, this generally isn't much of an issue. One thing that we did find to be an issue was that when our robot has red bumpers on, the LimeLight will be able to see part of the bumper and may report it as a note. To fix this, we made the bottom part of the camera feed black using `cv2.rectangle`:

```
image = cv2.rectangle(image,\
        (0, FOV_HEIGHT_PIX - BUMPER_HEIGHT_PIX),\
        (FOV_WIDTH_PIX, FOV_HEIGHT_PIX), (0, 0, 0), -1)
```

4 AprilTags

Apriltags act as fiducials that can help us navigate the field. This means, that we can now use a vision system to complement our odometry and get closer to accurate global positioning.

The First step was to build the AprilTag library on our coprocessor. Once it was built, we started programming using the python bindings. The most important part of the library is the Detector class. The Apriltag detector can be instantiated with several parameters including maxHamming distance and blur that may affect the quality of your detections. In our code, we made the maxHaming parameter equal to 0 and the blur parameter equal to .1. With these parameters we have no phantom tags, making our detections extremely robust.

4.1 Getting Camera Feed

OpenCV provides a good interface to get feed from cameras. Modifying that feed is not as intuitive. we want to set our frame to be a specific resolution, and also a specific framerate. We are able to do this only **AFTER** the video source is changed to MJPG. We also want to consider how a camera could be momentarily unplugged (God forbid), so we put it all in a loop.

```
def waitForCam(path):
    """Waits until there is a camera available at `path`.
    This is to ensure that cameras that are unplugged can be
    plugged back in and not interrupt the script."""
    while True:
        cap = cv.VideoCapture(path)
        cap:cv.VideoCapture
        cap.set(cv.CAP_PROP_FOURCC, cv.VideoWriter_fourcc('M', 'J', 'P', 'G'))
        cap.set(cv.CAP_PROP_FRAME_WIDTH, 1280)
```

```
        cap.set(cv.CAP_PROP_FRAME_HEIGHT, 720)
        cap.set(cv.CAP_PROP_FPS, 20)
        if cap.isOpened():
            print("open")
            return cap
        else:
            sleep(0.001)
```

4.2 Apriltag Detectors

Last year, we built the apriltag library from source. Unfortunately, we broke this installation when updating Linux on the computer. Now, we are using `pyapriltags`. The library required minimal refactoring from the previous year, and works just as fine! When we instantiate our `Detector`, we can put in a few helpful attributes. Increasing the quad_decimate parameter will increase the speed of the detector at the cost of detection distance. If you have extra cpu cores to throw at the problem then you can increase nthreads. If your image is somewhat noisy, increasing the quad_sigma parameter can increase speed.

```
def getDetector():
    """Returns an Apriltag Detector"""
    aprilobj = pyapriltags.Detector(
        nthreads=constants.NTHREADS,
        quad_decimate=constants.QUAD_DECIMATE,
        quad_sigma = constants.QUAD_SIGMA,
        refine_edges = constants.REFINE_EDGES,
        decode_sharpening= constants.DECODE_SHARPENING
        )
    return aprilobj

def getDetections(detector, frame):
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
    return detector.detect(gray)
```

One thing we found was that the detector would try to "clip" the edges of tags when the corners were off frame. This **severely** lowers the quality of the poses, so to counteract this, we set a check for each corner to determine it is at least a set margin away from the edge of the frame.

```
def allGoodCorners(l:list, framewidth:int, frameheight:int, margin:int) -> bool:
    """Detects corner clipping at the corner of a frame.
    Invalidates corners that are `margin` pixels away from the corner."""
    for corner in l:
        x, y = corner
        if x < margin
        or x > framewidth - margin
        or y < margin
```

```
        or y > frameheight - margin:
            return False
    return True
```

4.3 Multithreading

In case you didn't read the "Cameras" section, we have multi threaded the pose estimation process to have each of our 6 cameras run in parallel. First, we tried to differentiate each camera with their ID. However, this ID changes depending on the order you plug the devices. We needed something more robust. This is where the USB ports come in. We differentiate each of the cameras by the path of their USB port in `/dev/v4l/by-path`. It is critical to differentiate them as each lens has different intrinsic camera parameters that are integral to pose estimation in the following section.

On the default configuration of OpenCV, you cannot display more than one camera at once. We handle this by manually deciding which camera to view when we are running the non-headless process. View the following code:

```
valueLog = {}

def process_frame(
    cameraid,
    path,
    nt,
    headless = False,
    show_select = False,
    mainThreadLog:dict = {}
    ):
    ...

t6 = threading.Thread(target=process_frame, args=[
    10,
    os.path.realpath("/dev/v4l/by-path/pci-0000:06:00.3-usbv2-0:2:1.0-video-index0"),
    nt,
    headless,
    True,
    valueLog
])

cam_lst = [t1, t2, t3, t4, t5, t6]

print("starting cams")
for cam in cam_lst:
cam.start()
```

Here, `process_frame` is a function that all cameras run, and receives the id, path, an instance of networktables, a headless indicator, an indicator to show

the frame, and a logging dictionary that is shared between the threads. Each camera is responsible for logging its own data.

```
def logPose(camid, rx, ry, rt, fps, time, path=constants.LOG_PATH):
    """Logs camid, rx, ry, rt, and time of a pose."""
    with open(path, "a+", newline="") as log:
        c = csv.writer(log)
        c.writerow(
            [camid, rx, ry, rt, fps, time]
        )
```

4.4 Pose Estimation

Picture yourself walking in an amusement park and getting lost. You have a general idea of where you are based on how long you have been walking in a general direction but have insufficient information to pinpoint exactly where you are. Thankfully, there are several maps placed strategically throughout the park for this very purpose. You see the big red roller-coaster, recognize your location in relation to it, and now know to turn left after the photo booth to conclude your eventful day. Fiducials work the same way. Each feature of a fiducial has known 3D coordinates, allowing you to estimate your pose (position and orientation) in the world if you know your position relative to the fiducial. With pose information playing a critical role in the design of many systems, Fiducials have been developed in several research labs around the world, and have become a beacon of reliability to complement odometry systems.

4.4.1 Camera Calibration

The algorithm for deriving a pose from N known points is called the PnP1 problem. In order to use the PnP algorithm, you must first derive your intrinsic camera parameters. These parameters become the basis of pose calculations. Camera calibration is the process of identifying intrinsic camera parameters. Knowing 3D-2D correspondences tells you nothing if you don't know exactly how those 3D points are projected by your camera. Camera pose can be calculated by knowing two intrinsic camera parameters: distortion coefficients, and the camera matrix. These parameters define exactly how world coordinates are projected onto the image plane.

Projecting onto an image plane is very intuitive. Dr. Satya Mallick, who received his Ph.D. from the University of California San Diego, and who is also the current CEO of OpenCV, provides an excellent explanation in an article on learnopencv.com titled "Geometry of Image Formation." Consider a point in 3D space, the origin of which is placed arbitrarily. We have a camera at point Oc with a rotation and translation represented by R and T respectively. In our case, the x-axis points to the right, the y-axis points down, and the z-axis points out the front of the camera. We want to get point P in terms of the coordinate system of the camera. In order to do this, we multiply matrix R augmented

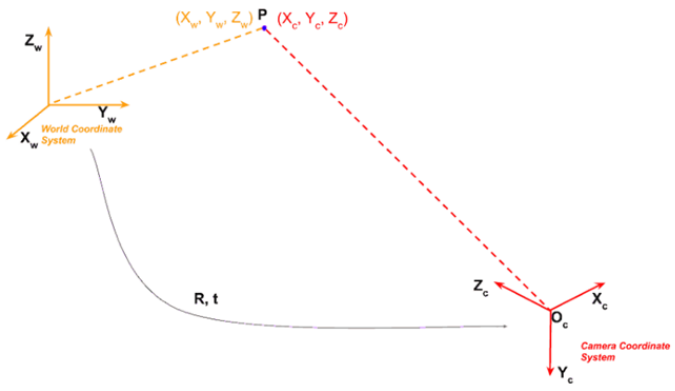


Figure 1: Transforming

with T, denoted by $[R|T]$ by the world coordinates of point P. For example, the resulting transformation matrix for a camera that has a rotation of 30 degrees about the Y axis relative to the world coordinate system is shown below.

$$[R|T] = \begin{bmatrix} \cos(30°) & 0 & -\sin(30°) \\ 0 & 1 & 0 \\ \sin(30°) & 0 & \cos(30°) \end{bmatrix}$$

In general:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_c = [R|T] * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w$$

With the point in the camera's coordinate system we can now project it onto the image plane. The distance the image plane is from the camera origin is the focal length. With our knowledge of similar triangles, we can solve for coordinates x,y with

$$x = f_x \frac{X_c}{Z_c}$$

$$y = f_y \frac{Y_c}{Z_c}$$

The different focal lengths for the x and y axes are due to the fact that the camera's horizontal and vertical focal lengths may not be the same resulting in non-square pixels. Now, there are also other factors that need to be considered. What if there is a skew between the plane and the sensor? What if the optical center does not lie directly in front of the sensor? We can implement all these changes in the following matrix. To get the image points, we simply divide the

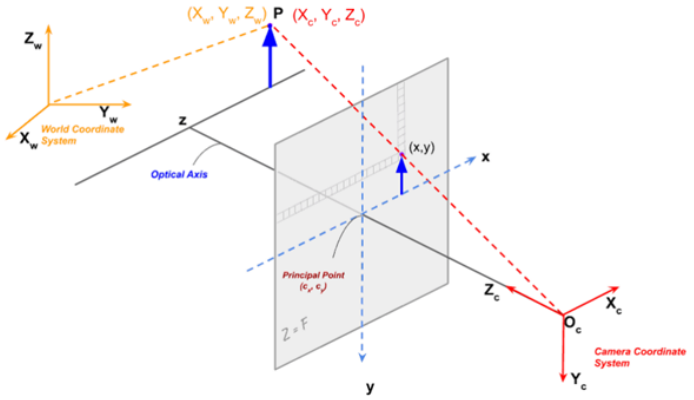


Figure 2: Projecting

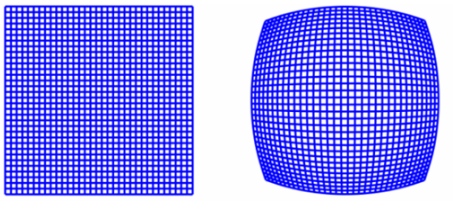


Figure 3: Radial Distortion

resulting x and y by z.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_c$$

$$X_{img} = x/z$$

$$Y_{img} = y/z$$

This is the intrinsic camera matrix. It represents all the camera parameters we need in order to project an image onto a camera plane. However, another factor impacts our projections: distortion. OpenCV describes two types of distortion: radial and tangential. Radial distortion is the type of distortion that results in the barrel effect. Radial distortion becomes larger the farther points are from the center of the image. Straight lines appear "bulged" out.

Radial distortion can be represented by the equation below:

$$x_{distorted} = x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$

$$y_{distorted} = y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$

On the other hand, tangential distortion occurs when there is a skew between the image-taking lens and the image plane. Tangential distortion can be represented by the equation below:

$$x_{distorted} = x + \left(2p_1xy + p_2\left(r^2 + 2x^2\right)\right)$$

$$y_{distorted} = y + \left(2p_2xy + p_1\left(r^2 + 2y^2\right)\right)$$

Basically, along with the intrinsic camera matrix, we need to find the distortion coefficients, k1, k2, p1, p2, and k3. OpenCV has a camera calibration tool that does this for us. We take pictures of a chessboard pattern so that it can generate the constants.

Now, we keep this constant in our `constants.py` file as so:

```
CAMERA_CONSTANTS = {
0: {
    "matrix":np.array([
    [1.1116882438499686e+03, 0.0, 6.4491686786616367e+02],
    [0.0, 1.1116882438499686e+03, 3.4494788214486431e+02],
    [0.0, 0.0, 1.0]
    ]),
    "distortion":np.array([
        [
        7.1115318392158552e-02,
        -9.0295426044896640e-02,
        0,
        0,
        0
        ]
    ]),
    "yc": .2965, #Y coordinate to center of robot.
    "xc": -0.0516, #X coordinate to center of robot.
    "thetar": 120 # Rotation to center of robot.
},
```

4.4.2 Perspective N Point

The Perspective and Point problem estimates the pose of a camera given 3D-2D correspondences and intrinsic camera parameters. We have all the tools we need. The fiducials we defined earlier in this paper, AprilTags, give us the 3D-2D correspondences, while our camera calibration gives us the intrinsic camera parameters. The OpenCV library has a function that performs the PnP algorithm called solvePnP. Therefore, we have all the tools necessary to solve the PnP problem. In order to use the values returned, we must first understand the math behind deriving position and angle. Our pose is represented by matrix $[R|T]$, a rotation matrix augmented by a translation vector. We get the components of this matrix from OpenCV's solvePnP function, which takes in object

points, image points, and the intrinsic camera parameters. For our purposes, we will input the real-world coordinates of the corners of our AprilTag, the 2D coordinates of the corners on the image frame, the camera matrix, and the distortion coefficients. solvePnP will undistort the image, and then solve for a 3x1 rotation vector and a 3x1 translation vector.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}_c = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} * [R|T] * \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w$$

The rotation vector can be turned into a rotation matrix using OpenCV's Rodrigues function. The transpose of said matrix augmented by the translation vector gives us [R—T]. But what do these components mean? The translation vector represents the translation from the origin of the camera's coordinate system to the origin of the world coordinate system. We want the inverse of that: the translation from the origin of the world coordinate system to that of the camera. We can invert this translation by negating the translation vector. Essentially, we are shifting our coordinate system the same amount from the origin but in the opposite direction. The same concept applies to rotation. Even after negating the values in the translation vector, the translation it represents is still with respect to a rotated coordinate system. We need to invert the rotation. We could do this by simply negating the angle, but that is the unknown we are solving for. Therefore, we invert the rotation by transposing the rotation matrix. After that, we multiply the transposed rotation matrix by the negated translation vector, giving us the coordinates of our camera:

```
mmat, rvec, tvec = cv.solvePnP(
    objectpoints,
    cornerpoints,
    cmtx,
    dist,
    )

tvec = (np.array(tvec))
rvec = (np.array(rvec))

rotationmatrix, _ = cv.Rodrigues(rvec)

"GETTING THE COORDINATES OF CAMERA"
final_coords = np.dot(-rotationmatrix.T, tvec)
#Multiply the negative inverse of the rotations to the translation vector.
```

Determining angle requires a bit more work. The rotation matrix that we have is a 3x3 matrix of floating-point numbers which does not give us any information on exact angles. We can extract angles from this rotation matrix by multiplying the rotation by a unit vector and projecting the rotated point

onto an axis depending on the angle you are solving for. For example, to get the angle rotated around the z-axis, we need to project the rotated vector onto the x-y plane:

$$V = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = [R] * V$$

$$\theta_z = \arctan\left(\frac{x}{y}\right)$$

In the code, it looks like this:

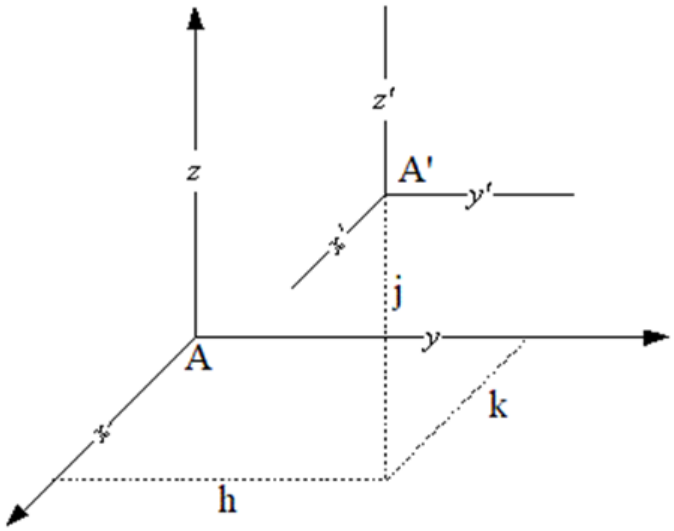
```
"GETTING THE ZTHETA (yaw of CAMERA on ground)"
# Get coordinates of rotated point on unit sphere.
# We want to project it onto the x-y axis
pointCoords = np.dot(rotationmatrix.T, np.array([[1],[0],[0]]))
pointX, pointY = [pointCoords[0][0], pointCoords[1][0]]

# Signs of x and y coordinates on unit circle
sx = 1 if pointX <= 0 else -1
sy = 1 if pointY <= 0 else -1
# # Modify theta based on coordinate quadrant to compensate
# for arctan only going from -90 to 90
ztheta = math.degrees(math.atan(pointCoords[1][0]/pointCoords[0][0]))\
        + (180*sy)*(sx - 1)/(-2)
ztheta -= 90
if ztheta < 0: ztheta += 360
```

Our camera now has a known position and orientation that increases. This yields highly accurate estimates, with angle and position accurate to a degree and a centimeter respectively. It is also important to note that the least squares fit performed by the PnP algorithm increases in precision as more points are passed in. Although the PnP algorithm is incredibly reliable for pose estimates, it only gives the pose of our camera. We want to know the pose of the robot.

4.4.3 Transforming Camera Pose to Robot Pose

Pose translations make use of the same fundamental idea as PnP: if I know where an object is, and I know where a second object is relative to that first object, I know where the second object is. In fact, we implemented this idea in the second and third sections of this paper. Using a transformation matrix, we put a point that was in the world's coordinate system into the camera's coordinate system, and we were even able to do that math backward and solve for the individual transformations.



Consider global coordinate system A, and transformed system A'. In this case, there is no rotation between A and A', meaning that we can do a simple translation. For any point P in coordinate system A, P in coordinate system A' can be represented as:

$$P_{A'} = P + \begin{bmatrix} -h \\ -k \\ -j \end{bmatrix}$$

The rule of thumb is to always pass in the values of your native coordinate system relative to the one you are translating to. That is why our values are negative. Applying this idea to a rotation, we can turn coordinate system A' θ degrees around the Z axes, by augmenting the matrix we had previously:

$$P_{A'} = \begin{bmatrix} \cos -\theta & -\sin -\theta & 0 & -h \\ \sin -\theta & \cos -\theta & 0 & -k \\ 0 & 0 & 1 & -j \end{bmatrix} \begin{bmatrix} P \\ 1 \end{bmatrix}$$

Now, lets apply this to our robot. Our "translated" coordinate system is our camera, with the x axis pointing forward, and the y axis pointing to the left. If the camera, relative to the world has coordinates and angle x_c, y_c, θ_c, and the center of robot in reference to the camera's coordinate system has coordinates h and k, we can find out the coordinates of the robot in the world's coordinate system! (we don't care about the z axis here, so it looks a tad bit different)

$$\begin{bmatrix} R_x \\ R_y \\ 1 \end{bmatrix}_w = \begin{bmatrix} \cos\theta_c & -\sin\theta_c & x_c \\ \sin\theta_c & \cos\theta_c & y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h \\ k \\ 1 \end{bmatrix}$$

For angle, we just add the yaw of the camera from earlier. Implementing both of these in code:

```
def getRobotVals(ay, cameraid, px, py):
    """Return positioning data of the robot. (rx, ry )"""
    robotheta = math.radians(ay - constants.CAMERA_CONSTANTS[cameraid]["thetar"])
    if robotheta > math.pi: robotheta -= math.tau
    xr = constants.CAMERA_CONSTANTS[cameraid]["xc"]
    yr = constants.CAMERA_CONSTANTS[cameraid]["yc"]

    transformationmatrix = np.array([
            [math.cos(math.radians(ay)), -math.sin(math.radians(ay)), px],
            [math.sin(math.radians(ay)), math.cos(math.radians(ay)), py],
            [0, 0, 1]
        ],
        dtype=object
    )
    robotcoordsRelativetocam = np.array(
        [
            [xr],
            [yr],
            [1]
        ],
        dtype=object
    )
    robocoords = np.dot(transformationmatrix, robotcoordsRelativetocam)
    return robocoords, robotheta
```

Wonderful! We have pose estimation!

4.5 Network Tables

Each camera sends over its own network table values. It will send rx, ry, rtheta, fps, time, and the number of tags it sees. Here is the implementation:

```
def networkConnect() -> any:
    """Copied from documentation. Establishes a connection to 10.6.39.2"""
    cond = threading.Condition()
    notified = [False]

    def connectionListener(connected, info):
        print(info, '; Connected=%s' % connected)
        with cond:
            notified[0] = True
            cond.notify()

    nt.initialize(server=constants.SERVER)
    nt.addConnectionListener(connectionListener, immediateNotify=True)

    with cond:
```

```
            print("Waiting")
            if not notified[0]:
                cond.wait()
    return nt

def pushval(networkinstance, tablename:str, theta, rx, ry, ntags, fps, time):
    """Pushes theta, rx, ry, ntags, and time values to the networktable.
    TableName should be the CameraID"""
    if networkinstance is None:
        return
    table = networkinstance.getTable(tablename)
    table.putNumber("theta", theta)
    table.putNumber("rx", rx)
    table.putNumber("ry", ry)
    table.putNumber("ntags", ntags)
    table.putNumber("fps", fps)
    table.putNumber("time", time)
```

4.6 Deploying

To get the processes to begin on startup, we created a new systemd process on our vision PC.

```
cd /etc/systemd/system
Touch vision.service
Vim vision.service
```

Now, in `vision.service` set `user`, `Working Directory`, `ExecStart`, `wanted_by`, and `Description`. Here is what team 639 (us!) did:

```
[Unit]
Description="Vision Process"

[Service]
User=crr
WorkingDirectory=/home/crr/2024Visions
ExecStart = /bin/python3 /home/crr/2024Visions/pose_estimation_pyapriltags.py

[Install]
WantedBy=multi-user.target
```

5 Driving to the Note (and Other Places)

When we saw the game, we immediately knew that we would have to implement a system to effectively drive to the note, as there are countless obstructions in the view of the driver. The limelight passes a translation to the roborio via

networktables, and we use those translations to generate a path to the note while running our acquisition. It works all the time.

6 Auto-Aim

To score in both the amp and speaker, we use the pose from the pose estimation process to align ourselves. We take our current pose, and find the transformation that we need to be either facing the middle of the speaker, or move to the correct position to score into the amp.